TP4 - Héritage, classes abstraites

m billaud

26-02-2021

Table des matières

Dans ce TP on va utiliser des classes, des interfaces et des classes abstraites pour modéliser des **cadeaux de Noël**.

Les cadeaux sont de différentes sortes (livres, abonnements, etc), ils peuvent être emballés et groupés dans des boites.

1 Polymorphisme

Le code suivant

```
Livre livre = new Livre("Les misérables");
Abonnement abonnement = new Abonnement("Club de belote");
Cadeau[] cadeaux = {
    livre,
    abonnement
};
for (var c : cadeaux) {
    System.out.println("-> " + c.toString());
}
```

doit afficher

```
-> le livre "Les misérables"
-> un abonnement pour Club de belote
```

Pour cela

- définissez une interface `Cadeau`, implémentée par deux classes `Livre` et `Abonnement`
- redéfinissez (overriding) les méthodes `toString` des deux classes.

2 Classe abstraite

Les classes `Abonnement` et `Livre` ont des points communs : elles ont un attribut (initialisé par le constructeur) qui décrit la nature du cadeau ("Les misérables", ou "Club de belote").

Modifiez les deux classes pour qu'elles dérivent d'une (nouvelle) classe abstraite qui détiendra cet attribut.

```
   Cadeau           Interface
     ^
 CadeauAbstrait     Classe abstraite (attribut, constructeur)
  ^     ^
Livre   Abonnement  Classes (constructeur, toString)
```

3 Circularité

Le code

```
Cadeau c1 = new CadeauEmballe("du papier bleu", livre);
System.out.println(c1);
```

doit produire

```
le livre "Les misérables" emballé dans du papier bleu
```

Ceci introduit une circularité dans le modèle de données : la classe `CadeauEmballe` 1) est dérivée de `Cadeau` 2) a un attribut de type `Cadeau` :

```
           Cadeau    <--\
             ^            |  contient un
dérive de    |            |
      CadeauEmballe  ---/
```

- Implémentez la classe `CadeauEmballe`
- une méthode utilitaire statique (un “helper”) simplifiera les écritures

```
Cadeau c2 = emballer(livre, "du papier bleu");
Cadeau c3 = emballer(c2, "du papier kraft");
```

4 Composite

Un cadeau peut aussi être une `Boite`, contenant d’autres cadeaux

```
  Cadeau  <---\
  ^    ^       |  contient des
 ...   |       |
     Boite ---/
```

4.1 La classe `Boite`

Nous voudrions que le code

```
Boite boite = new Boite(
                    emballer(livre, "du papier journal"),
                    emballer(abonnement, "un petit mot gentil")
            );
System.out.println(boite));
```

affiche un texte du style

```
une boite contenant le livre "les Misérables" emballé
dans du papier journal, un abonnement contenu dans un
petit mot gentil
```

Écrivez la classe `Boite`, avec

- un constructeur à nombre variable d’arguments (voir annexe) ;
- sa méthode `toString`

4.2 Autres méthodes

1. Une méthode “`afficherEmballages`” qui affiche tous les emballages utilisés dans un cadeau. Indication

- rien à faire pour un `Livre` ou un `Abonnement` ;
- pour un `CadeauEmballe`, afficher l’emballage utilisé, et appeler `afficherEmballages` sur son contenu ;
- appeler la méthode sur chacun des cadeaux contenu dans une `Boite` ;

2. Une méthode “`nombreDeBoites`” qui retourne ce que vous devinez.
3. Une méthode “`listeLivres`” qui retourne un `ArrayList<Livre>` des livres reçus. Indication : pour un `Livre`, c’est un `ArrayList` le contenant.

5 Annexes

5.1 Une méthode à nombre variable d’arguments

Exemple de méthode qui accepte un nombre variable de paramètres

```
static int somme(int... nombres) {
    int total = 0;
    for (var n : nombres) {
        total += n;
    }
    return total;
}
```

La notation “`int...`” indique que le paramètre reçu est un `int[]`, qui est construit lors de l’appel

```
int r =  somme(12, 34, 567);
```

dans un tableau temporaire rassemblant les paramètres :

```
int[] tmp = { 12, 34, 567 };
int r = new Boite(tmp);
```

5.2 Complément : méthodes par défaut

Plutôt que faire

```
Cadeau c2 = emballer(livre, "du papier bleu");
```

une alternative est d’utiliser

```
Cadeau c2 = livre.dans("du papier bleu");
```

en définissant une “méthode par défaut” dans l’interface `Cadeau` :

```
public interface Cadeau {

    default Cadeau dans(String emballage) {
        return new CadeauEmballe(emballage, this);
    }
}
```

En effet, depuis Java 8, les interfaces peuvent (sous certaines restrictions) posséder des méthodes en plus des signatures. Ce qui les rapproche des classes abstraites.

Avant Java 8, il aurait fallu transformer l’interface `Cadeau` en classe abstraite. C’est possible, mais ça en limite les possibilités d’emploi : une classe peut *implémenter* plusieurs interfaces, mais ne peut étendre qu’**une seule** classe.